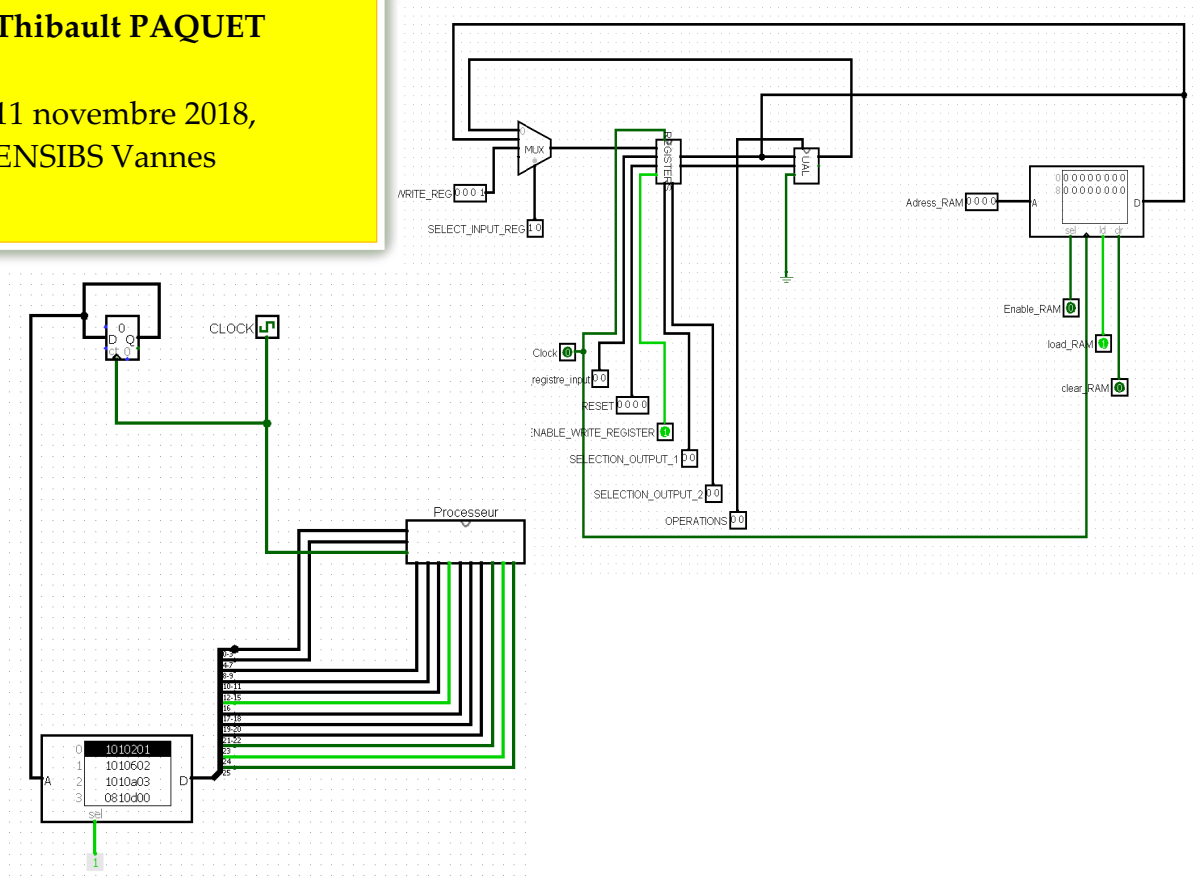
Rapport TP1
Signaux et systèmes
Conception d'un processeur simple

Étudiants :

Kevin MOREAU
Thibault PAQUET

11 novembre 2018,
ENSIBS Vannes



École d'ingénieurs
ensibs
Université Bretagne Sud

Introduction

Dans le cadre de notre première année en cyberdéfense en tant qu'apprentis alternants, notre objectif est de concevoir un processeur simple sous Logisim.

Nous aurons pour objectif, après avoir pris en main le logiciel, de concevoir chacun des composants constituants le processeur à savoir :

- *Les circuits logiques de base (AND, OR, …)*
- *L'unité arithmétique logique (UAL)*
- *Le banc de registres*
- *L'intégration d'une mémoire ROM*
- *L'intégration d'une mémoire RAM*

Une fois crées, ces composants devront être assemblées pour répondre à un jeu d'instructions d'abord établi manuellement, puis en provenance d'instructions basiques tapés dans un fichier texte. Ce fichier texte contenant les instructions sera alors converti pour être chargé puis exploité par notre processeur.

SOMMAIRE

1 – Prise en main du logiciel Logisim

2 – Conception d'un processeur simple

2.1 Conception des circuits logiques AND / OR / XOR (données de 4 bits)

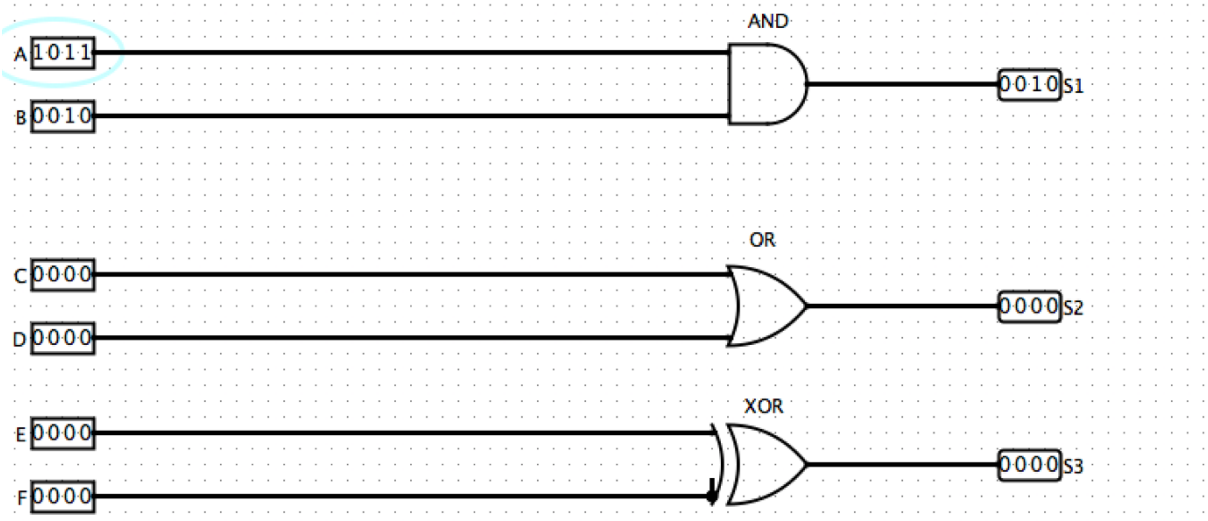


Schéma de 3 paires de données de 4 bits associées à des portes logiques AND / OR / XOR

2.2 Conception de l'additionneur

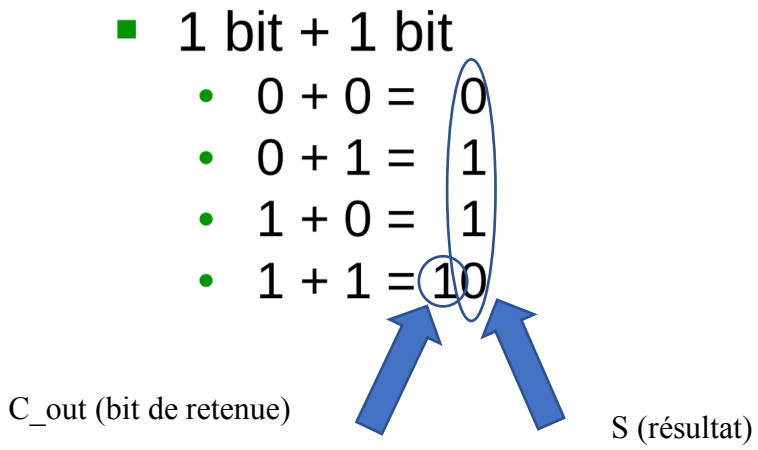
2.2.1 Conception d'un additionneur 1 bit + 1 retenue

Pour établir un additionneur 1 bit, il est important de comprendre le fonctionnement même d'une addition binaire :

- 1 bit + 1 bit
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 10

Il est important de souligner que la somme de deux bits se passe de cette façon, et qu'il existe un bit supplémentaire, que l'on va appeler C qui permet de tenir compte de la retenue entre deux bits égaux à 1 (en base2, la valeur maximale étant 1
La prise en compte de la retenue implique l'utilisation de deux sorties :

- La première, que l'on note S, permet de définir le véritable résultat de la somme des deux bits, et qui fonctionne sur le principe d'un XOR (OU EXCLUSIF) ; en effet, si l'on regarde bien, on se rend compte que la valeur de S se comporte comme un ou exclusif entre nos deux bits additionnés



- La deuxième que l'on note ici C_out doit être prise en compte lors de l'opération entre les deux bits. En effet, si l'on reprend le cas précédent, lorsque nous avons au moins 2 bits égaux à un lors de l'addition, une retenue sera systématiquement présente.

Ainsi pour créer notre additionneur 1 bit, nous allons établir la table de vérité suivante :

A	B	C_in		S	C_out
0	0	0		0	0
0	0	1		1	0
0	1	0		1	0
0	1	1		0	1
1	0	0		1	0
1	0	1		0	1
1	1	0		0	1
1	1	1		1	1

On en déduit donc les équations logiques suivantes, respectivement pour S et pour C_out :

$$S = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$$

$$C_{out} = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

Afin de simplifier les équations ci-dessus et alléger le nombre de portes logiques au sein de notre circuit, nous dressons un Tableau de Karnaugh pour chaque sortie ci-dessous :

S	AB			
C_in	00	01	11	10
0		1		1
1	1		1	

Ici la simplification n'est pas très exhaustive, on peut cependant factoriser l'équation initiale en C :

$$S = \overline{AB}C + \overline{A}B\overline{C} + A\overline{BC} + ABC$$
$$S = \overline{C}(\overline{A}B + A\overline{B})$$
$$S = \overline{C}(A \oplus B)$$

Par contre, pour la deuxième équation, c'est ici que le tableau de Karnaugh va s'avérer très efficace :

C_out	AB			
	00	01	11	10
0			1	
1		1	1	1

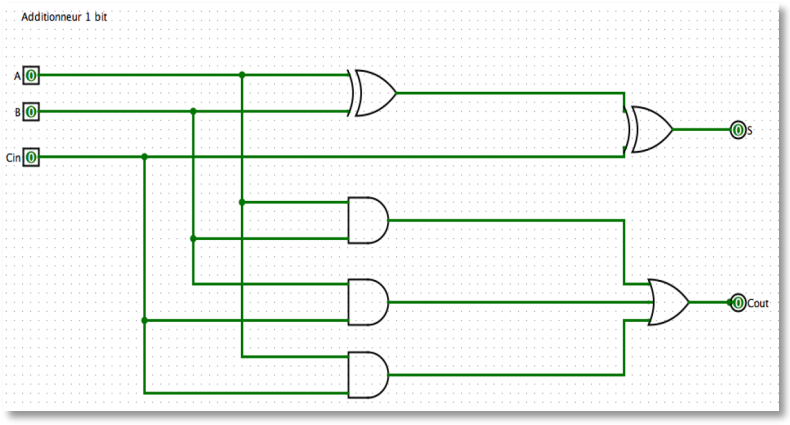
On applique les groupements :

C_out	AB			
	00	01	11	10
0			1	
1		1	1	1

En simplifiant, on en déduit l'équation suivante :

$$C_{out} = AB + BC + AC$$

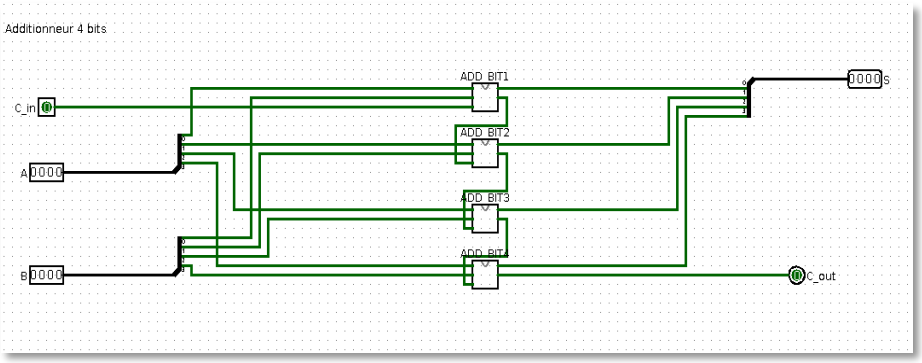
Nous obtenons enfin, le circuit suivant pour notre additionneur à 1 bit avec retenue :



2.2.2 Conception d'un additionneur 4 bits

Pour établir l'additionneur 4 bits on peut réutiliser l'additionneur 1 bit. A l'instar de notre additionneur 1 bit, l'additionneur 4 bits va recevoir deux entrées de 4 bits qui seront les valeurs à calculer et une entrée de 1 bit qui correspond au flag de retenue.

On passe chaque bit de nos valeurs à calculer à un additionneur 1 bit et la retenue du calcul du bit précédent (ou du flag Retenue pour le premier additionneur). L'additionneur nous retourne le résultat de l'addition des 2 valeurs de 4bits et la retenue de cette addition.



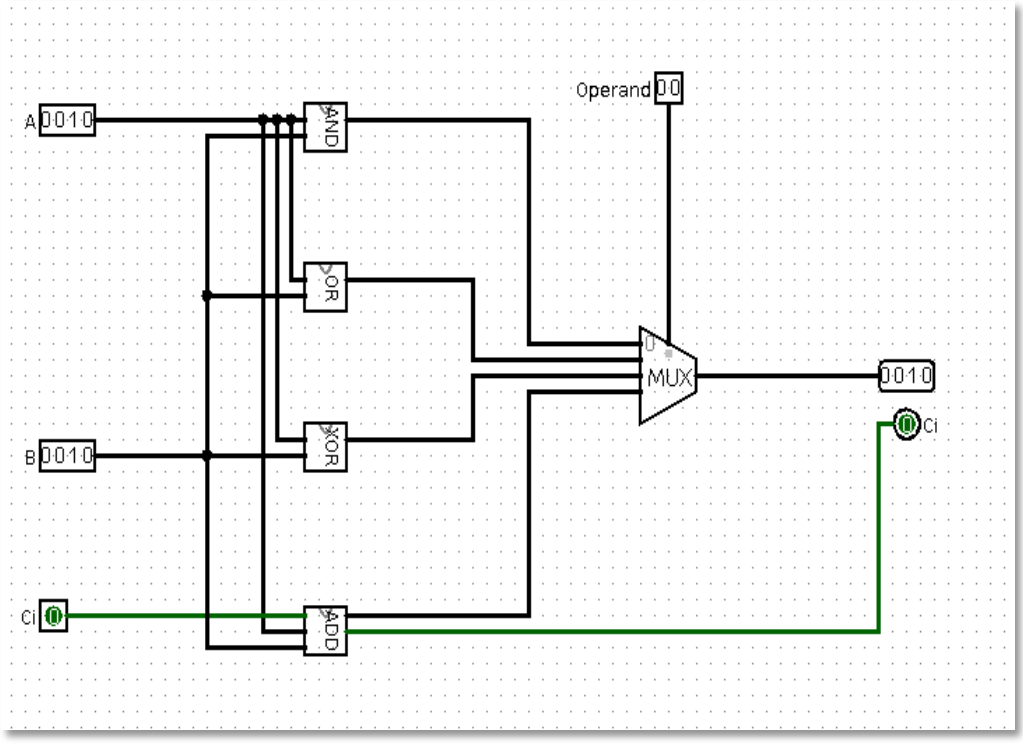
2.3 Conception de l'UAL

Une Unité d'Arithmétique Logique est la partie du processeur qui va effectuer les calculs entre des entrées. Le choix de l'opérateur est géré par un signal de commande spécifique pour chaque opérateur. Notre UAL va recevoir en entrée deux valeurs de 4 bits et le flag de Retenue. Il faut aussi que notre UAL nous retourne uniquement le résultat de l'opération choisit, pour cela on utilise un multiplexeur.

Le multiplexeur va recevoir les résultats des calculs des 4 opérations utilisé dans cette UAL (OR, AND, XOR, Addition) et sortir uniquement le résultat du calcul voulu selon les 2 bits du signal définissant l'opérateur (Operand).

Correspondance entre le signal Operand et l'opérateur :

AND	00
OR	01
XOR	10
ADDITION	11



Notre UAL donne en sortie le résultat de l'opération voulu et le flag Retenue s'il s'agit d'une addition.

2.4 Conception d'un banc de registres

2.4.1 Registre 4 bits

Un registre est un emplacement mémoire au sein du processeur. Pour réaliser notre registre il faut que l'état de la sortie soit maintenu même après la disparition du signal de contrôle.

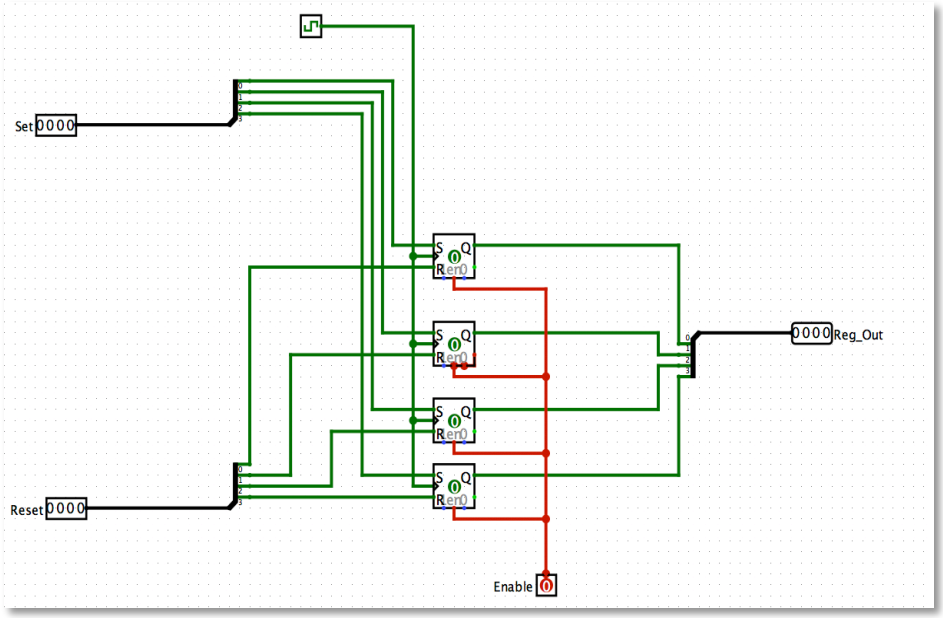
Pour cela nous pouvons utiliser des bascules. Il existe plusieurs types de bascules, pour notre processeur nous pouvons utiliser des bascules synchrones de type RS.

Ces bascules utilisent une horloge pour changer l'état de la sortie et sont utilisés pour stocker des données.

Pour stocker nos 4 bits d'entrée nous allons utiliser quatres bascules. Chaque bascule reçoit en entrée un bit de la valeur à stocker, le signal de l'horloge pour le changement d'état et retourne la valeur de sortie non-inversé et inversé.

Ces bascules disposent d'une entrée pour un signal de réinitialisation. Dans notre cas, celui-là sera codé sur 4 bits afin de pouvoir réinitialiser les 4 bascules 1 bit.

Enfin, chaque bascule dispose d'un signal d'activation correspondant à l'entrée « enable ». Ainsi, le registre de 4 bits sera actif dès lors qu'il recevra un signal via l'entrée « enable ».



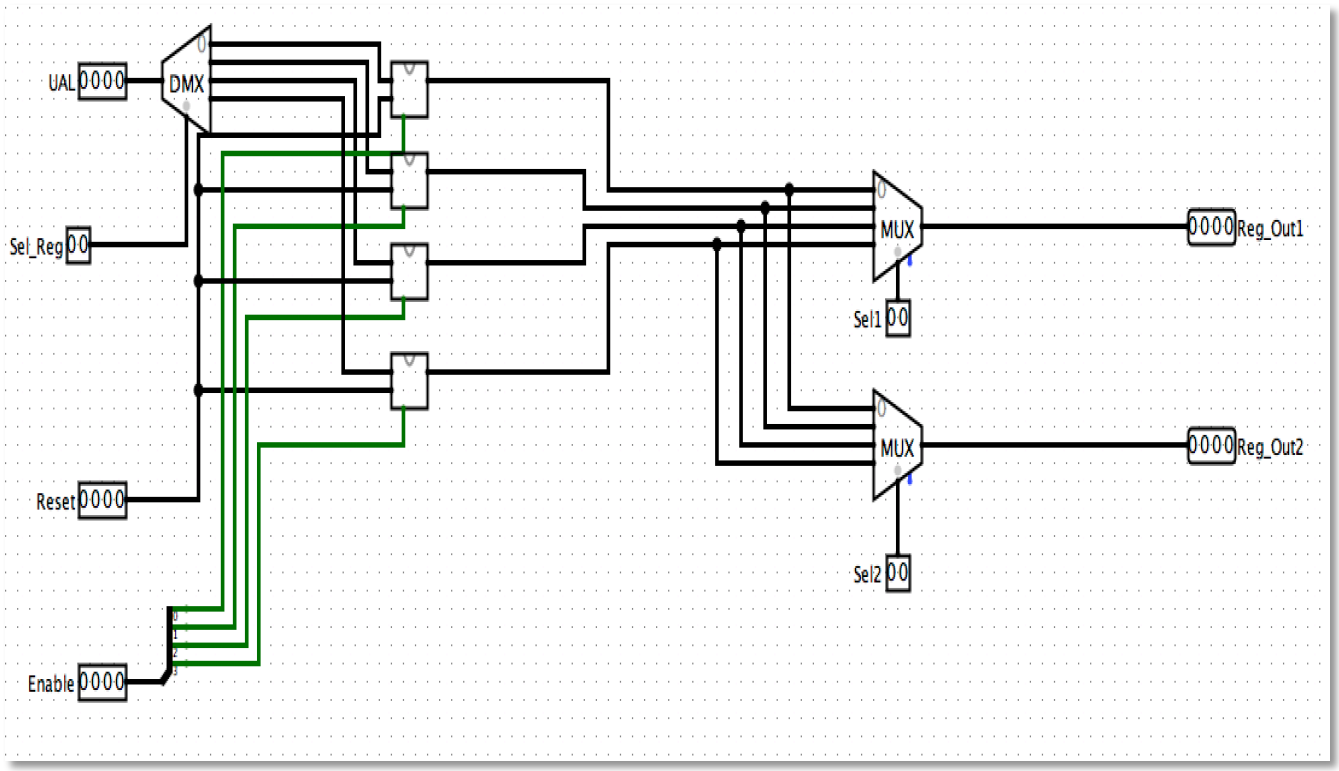
2.4.2 Intégration de 4 registres 4 bits

Pour réaliser les 4 registres de 4 bits nous utilisons les registres de 4 bits précédemment crées. On passe en entrée de chaque registre les bits à stocker.

L'horloge n'a pas besoin d'être intégrée ici car elle est interne à chaque registre 4 bits.

Par ailleurs, afin de prévoir le retour du signal provenant de l'UAL, on récupère ce dernier en entrée que l'on va démultiplier à chaque entrée du registre en utilisant un démultiplexeur.

Celui-ci a pour rôle de recopier le signal provenant de l'UAL à l'entrée de chaque registre et, selon les 2 bits de sélection choisis directement sur celui-ci, on va pouvoir attribuer le signal de l'UAL sur le registre de notre choix.
Enfin, en sortie, nous allons récupérer 2 valeurs des registres qui vont re-interagir avec l'UAL (ce dernier a besoin de deux signaux pour effectuer un calcul). On utilise pour cela deux multiplexeurs que l'on place en parallèle disposant chacun de deux bits de sélection pour choisir le registre que l'on souhaite utiliser.



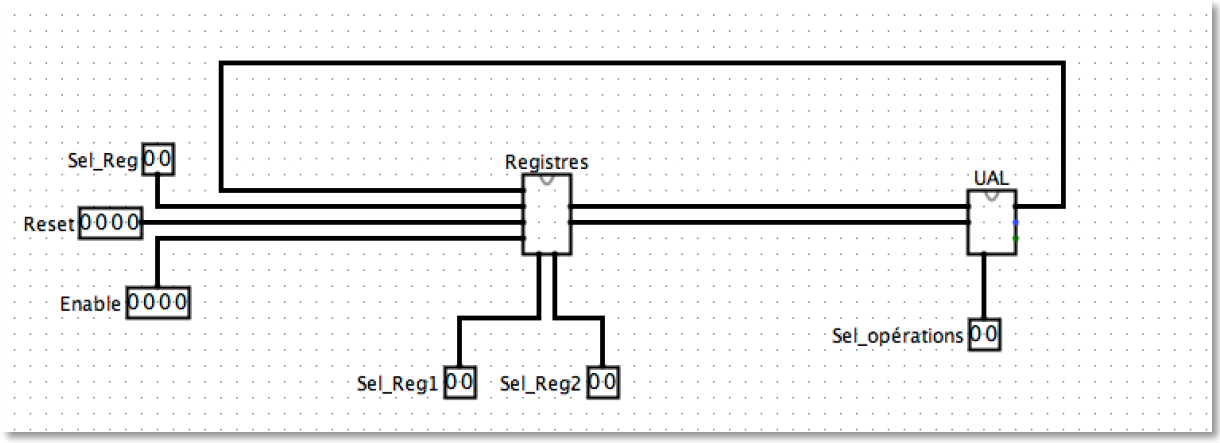
2.5 Intégration dans un processeur complet

Le but ici est d'intégrer chacun des éléments vus précédemment ensemble afin de créer la structure de base d'un processeur. On utilise ainsi le banc de 4 registres 4 bits que nous avons créés, l'UAL permettant de traiter les calculs et les différents bits de sélections et de réinitialisation nécessaire pour le système.

On retrouve ainsi :

- 2 bits de sélection de registres permettant de savoir vers quel registre l'opération résultant de l'UAL va être stockée : **Sel_Reg**
- 4 bits de réinitialisation où chacun des 4 bits de réinitialisation est distribué sur un registre 4 bits : **Reset**
- 4 bits « enable » où chacun des 4 bits d'activation est distribué sur un registre 4 bits. Ce même bit est ensuite distribué sur les bascules RS de chaque registre : **Enable**
- 2 x 2 bits de sélection des registres qui vont permettre de choisir les deux signaux en sortie du banc de registre qui vont être utilisé par l'UAL en entrée : **Sel_Reg1** & **Sel_Reg2**
- 2 bits de sélection qui vont permettre de décider de quel opération l'UAL doit positionner à sa sortie : **Sel_opérations**

Le fonctionnement est le suivant ; pour l'instant on insère des valeurs manuellement au sein des 4 registres en forçant les valeurs. Une fois ces valeurs inscrites, on sélectionne manuellement les 2 registres qui vont se positionner à l'entrée de l'UAL grâce aux deux sélectionneurs **Sel_Reg1** et **Sel_Reg2**. Une fois les deux valeurs en entrée de l'UAL, on sélectionne l'opération que l'on souhaite positionner en sortie de l'UAL grâce à **Sel_opérations**. Ensuite, la sortie est bouclée vers l'entrée du banc de 4 registres 4 bits. On choisit dans quel registre on souhaite placer ce signal en interagissant avec le démultiplexeur piloté par **Sel_Reg**. En parallèle bien entendu, le registre en question doit être activé (« **enable** ») pour que la valeur du signal provenant de l'UAL soit bien enregistrée dans le registre à chaque front d'horloge.



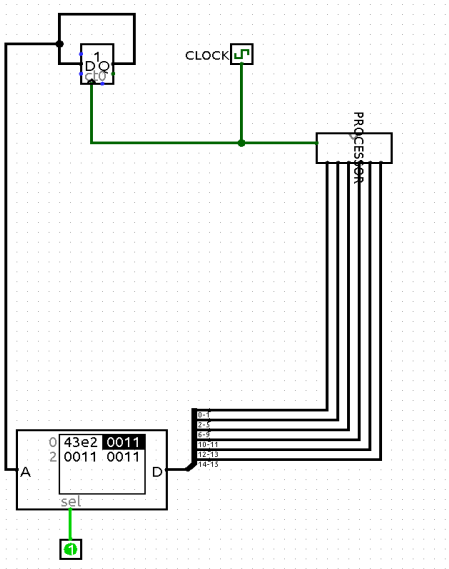
3. Conception d'un processeur 4 bits avec Pointeur Programme et mémoire d'instruction

3.1 Jeu d'instructions avec une mémoire ROM & ajout de la mémoire RAM

Le jeu d'instructions va contenir les actions à réaliser par le processeur. Chaque instruction est stockée à une adresse mémoire de la ROM et est noté en hexadécimal. Le bit de sélection *sel* permet d'activer la ROM. Sur le pin A on reçoit le pointeur vers l'instruction, géré par un compteur (voir partie suivante).

L'instruction va ensuite être transmise au processeur :

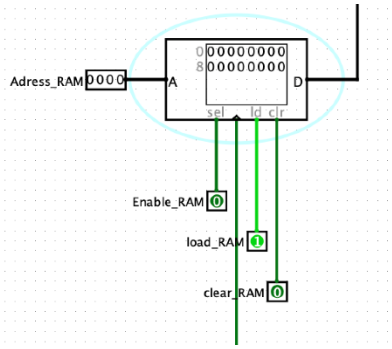
- 4 premiers Bits, le registre que l'on souhaite activer en entrée.
- 2 premiers Bits, le registre que l'on souhaite activer en entrée.
- 2 premiers Bits, le registre que l'on souhaite activer en entrée.
- 4 bits suivants, le reset pour chacun des registres.
- 4 bits suivants, activation en écriture du ou des registres.
- 2 bits suivants, la première valeur que l'on souhaite récupérer pour l'UAL.
- 2 bits suivants, la seconde valeur que l'on souhaite récupérer pour l'UAL.
- 2 derniers bits, l'opération que l'on souhaite effectuée.



Le compteur s'incrémente de 1 à chaque tour d'horloge, cela va permettre de parcourir les instructions contenues dans la ROM. Chaque valeur du compteur correspond à une adresse dans la ROM avec une instruction à cette adresse. On passe le pointeur à la ROM sur le pin A.

L'instruction est ensuite passée au processeur qui la réalise. Une fois l'instruction réalisée le compteur s'incrémente de nouveau et l'instruction suivante est réalisée.

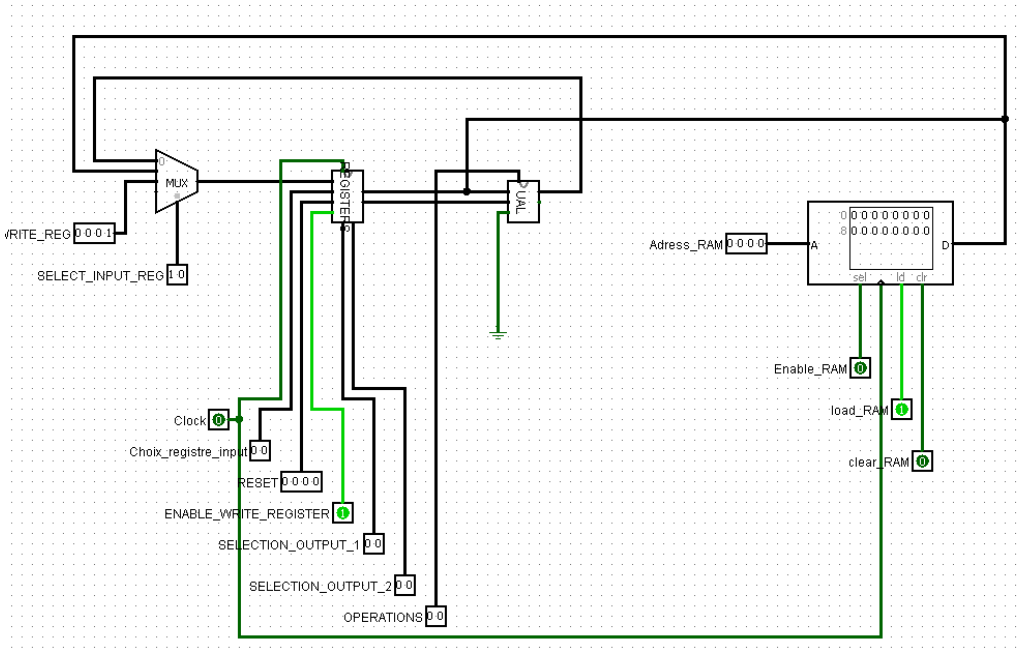
Concernant l'ajout de la mémoire RAM, nous l'avons implémenté directement dans le module du processeur :



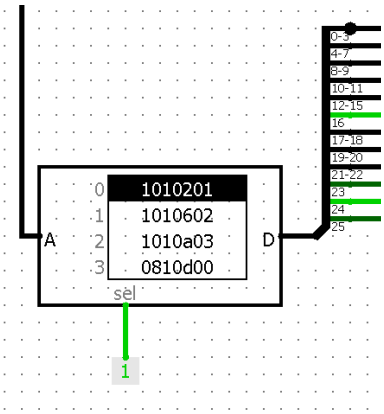
Celle-ci a besoin de différents paramètres pour fonctionner correctement :

- ***A*** : 4 bits permettant de déterminer l'adresse de la valeur que l'on veut stocker (ici maximum 16 adresses)
- ***D*** : La donnée à charger sur le bus ou à lire, la permutation entre les deux modes est gérée avec le pin « ***ld*** », qui, lorsqu'il est à 1, va récupérer les données entrant dans la ram.
  Lorsque ld est à 0, les valeurs seront chargées depuis la ram.

- ***Sel*** : Permet l'activation du composant
- ***Clr*** : Permet la remise à zéro des valeurs sauvegardées dans les adresses de la RAM.
- Ces différents composants sont ensuite gérés par les autres composants de notre processeur.



Par ailleurs, il est également important de signaler qu'avec l'ajout de ce composant, le nombre de bits d'instructions fournis par la ROM augmente.



Notre mémoire ROM possède un total de 26 bits de commande décomposée comme suit :

0000 0000 00 00 0000 0 00 00 00 0 0 0

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]

N°	Variable	Description
1	WRITE_REG	Valeur brute à insérer dans un registre
2	Adress_RAM	Permet de définir l'adresse utilisée dans la RAM
3	SELECT_INPUT_REG	Interaction avec le multiplexeur en entrée du banc de registres pour déterminer quelle entrée utilisée (00 : UAL ; 01 : RAM ; 10 : ROM ; 11 : N/A
4	Choix_registre_input	Interaction avec le démultiplexeur dans le banc de registres pour déterminer dans quel registre stockée la valeur en entrée (00 : R0 ; 01 : R1 ; 10 : R2 ; 11 : R3)
5	RESET	Permet de remettre à zéro l'ensemble des registres
6	ENABLE_WRITE_REGISTER	Permet d'envoyer un signal d'autorisation d'écriture sur un registre particulier. La sélection d'autorisation d'écriture du registre se fait via le même DMUX que Choix_registre_input
7	SELECTION_OUTPUT_1	Permet de choisir le premier registre qui sera en entrée de l'UAL (00 : R0 ; 01 : R1 ; 10 : R2 ; 11 : R3)
8	SELECTION_OUTPUT_2	Permet de choisir le deuxième registre qui sera en entrée de l'UAL (00 : R0 ; 01 : R1 ; 10 : R2 ; 11 : R3)
9	OPERATIONS	Permet de choisir quelle opération sera effectuée par l'UAL (00 : ET ; 01 : OU ; 10 : OU EXCLUSIF ; 11 : ADDITION)
10	Enable_RAM	Permet d'activer la RAM (composant)
11	Load_RAM	Si 1 : Charge la valeur dans la RAM vers la sortie Si 0 : Permettre l'écriture de données dans la RAM
12	Clear_Ram	Permet de remettre à zéro les données de la RAM

3.2 Mise en pratique avec instructions choisies par un utilisateur

Cette partie permet l'exploitation totale du processeur.
Selon les instructions saisies par l'opérateur dans un fichier texte, le processeur réagira en fonction de celles-ci pour fournir le résultat attendu par l'utilisateur.

Principe de fonctionnement :

1. L'utilisateur utilise un template de rédaction fourni et écrit ses instructions en fonction des règles établies :

```
LD R0, 1
LD R1, 2
LD R2, 3
LD R3, 4
ADD R0, R1, @4
LD R1, @4
OR R2, R3, R0
RST
```

2. Grâce à un script python, les instructions saisies par l'utilisateur dans ce fichier texte seront converties en binaire puis exportées en hexadécimal dans un fichier binaire.

Exemple avec une instruction :

Instruction	Conversion binaire	Conversion hexadécimale
LD R0, 1	00010000000000001000000000	4194816

3. L'utilisateur charge ensuite le binaire dans la mémoire ROM

4. A chaque coup d'horloge, la mémoire ROM passera d'une instruction à une autre.

Conclusion

Le processeur crée ici prend donc en compte un certain nombre de fonctions :

- *L'insertion de valeur via la ROM directement depuis un registre*
- *Opérations de deux valeurs entrant dans l'UAL puis résultat enregistré dans un registre.*
- *Déplacement de valeur depuis le registre vers la RAM*
- *Déplacement d'une valeur de la RAM dans un registre*
- *Remise à zéro des registres*
- *Autorisation d'écriture propre à chaque registre*

Concernant nos axes d'améliorations, nous aurions bien voulu un peu plus pousser les fonctionnalités comme par exemple l'insertion d'une valeur dans la RAM depuis la ROM sans passer par le banc de registres.

Par ailleurs, dans le cadre de ce TP, le nombre d'instructions est limité ici à 8 pour des raisons évidentes de test et d'exploitation par un humain. Nous aurions très bien pu limité ces instructions à un nombre beaucoup plus grand.

Enfin, ci-dessous, les fichiers annexes utiles au bon fonctionnement de notre processeur.

Fichiers annexes

1. Asm.txt : Fichier texte dans lequel seront inscrits les instructions de notre programme
2. Asm_to_bin.py : Petit script permettant de convertir le fichier asm.txt en un fichier asm.bin exploitable par le processeur

Commande à réaliser : python3 asm_to_bin.py asm.txt

3. Instructions.txt : Fichier texte contenant les instructions nécessaire à la bonne écriture du fichier asm.txt

asm.txt

asm_to_bin.py

instructions.txt